CHAPTER 3

Creating the Project

In this chapter, I show you how to create a project that contains ASP.NET Core MVC and Angular applications, which means both parts of the project can be developed using Visual Studio or Visual Studio Code. This project forms the foundation for the rest of this book, as I explain how to use Angular and ASP.NET Core MVC to create a rich web application. Table 3-1 puts the combined project in context.

Tip You can download the complete project for this chapter from `https://github.com/apress/esntl-angular-for-asp.net-core-mvc`. This is also where you will find updates and corrections for this book.

Table 3-1. *Putting a Combined Project in Context*

Question	Answer
What is it?	A combined project includes Angular and ASP.NET Core MVC in a single folder structure.
Why is it useful?	A combined project makes it easy to develop both parts of an application using a single IDE, such as Visual Studio, as well as simplifying the process of using an ASP.NET Core MVC web service to provide data to Angular.
How is it used?	The Angular application is created first, followed by ASP.NET Core MVC. Additional NuGet packages are used to allow both parts of the project to work together at runtime.
Are there any pitfalls or limitations?	A combined project makes managing the development process easier, but you still need a good working knowledge of both Angular and ASP.NET Core MVC to create an effective application.

Are there any alternatives?	You can develop the Angular and ASP.NET Core MVC parts of the application separately, although this tends to complicate the development process.

Preparing to Create a Project

There are several different ways to create a project that combines Angular and ASP.NET Core MVC. The approach that I use in this book relies on the @angular/cli package, used in conjunction with the .NET tools for creating a new MVC project.

The @angular/cli package provides a command-line interface that simplifies the process of creating and working with a new Angular project. During development, the Angular code is compiled, packaged, and delivered to the browser using webpack, a popular tool for creating JavaScript bundles that contain only the code required to run a project.

To create a project that combines Angular and ASP.NET Core MVC, I start with @angular/cli and access the underlying webpack configuration, which can then be used to integrate the Angular tools and libraries into the ASP.NET Core project.

To start this process, open a new command prompt and run the command in Listing 3-1 to install the @angular/cli package. If you are using Linux or macOS, you may need to use **sudo** to run this command.

Listing 3-1. Installing the @angular/cli Package

```
npm install --global @angular/cli@1.6.6
```

This command takes a while to run because the package has a lot of dependencies, all of which have to be downloaded and installed.

Creating the Project

The process for creating a combined Angular and ASP.NET Core MVC project requires careful attention: both development platforms have their tools and conventions, and getting them to work together requires a specific set of steps to be performed in order. In the sections that follow, I walk through the process and explain each step.

Tip You must follow each step exactly as shown, without missing a step or changing the order. If you get stuck, then you can download a ready-made project from the source code repository, which is linked from the Apress.com page for this book.

Creating the Angular Part of the Project

The first step is to create a new Angular project, which is done using the @angular/cli package installed in Listing 3-1. Open a new command prompt, navigate to the folder where you keep your development projects and run the commands shown in Listing 3-2.

Listing 3-2. Creating an Angular Project

```
ng new SportsStore --skip-git --skip-commit --skip-tests
```

The `ng` command is provided by the @angular/cli package, and `ng new` creates a new Angular project. The arguments that start with `--skip` tell @angular/cli not to perform some of the standard setup steps that are usually included in projects, and the `--dir` argument specifies the name of the folder that will contain the source code for the Angular application. The `SportsStore` folder will contain the complete project - including the ASP.NET Core MVC part of the application - and the Angular code lives in a folder called `ClientApp`.

The `ng new` command in Listing 3-2 creates a folder called `SportsStore` that contains the tools and configuration files for an Angular project, along with some placeholder code so that the tools and project structure can be tested.

The rest of the setup is performed inside the project folder, so run the command shown in Listing 3-3 to change the working directory and rename the folder that contains the client-side part of the application.

Listing 3-3. Changing the Working Directory

```
cd SportsStore

mv src ClientApp
```

Getting the Angular tools to work with the .NET tools requires additional NPM packages. Run the commands shown in Listing 3-4 to install these packages.

Listing 3-4. Adding NPM Packages to the Project

```
npm install --save-dev aspnet-prerendering@3.0.1

npm install --save-dev aspnet-webpack@2.0.3

npm install --save-dev webpack-hot-middleware@2.21
```

Microsoft provides some of these NPM packages, and they are used to set up and run the Angular development tools inside the ASP.NET Core runtime. These packages work directly with webpack, which is usually hidden when working with projects created using @angular/cli.

Open the `.angular-cli.json` file and change the configuration setting shown in Listing 3-5 to ensure that the project settings reflect the `mv` command from Listing 3-3.

Listing 3-5. Changing the Project Configuration in the .angular-cli.json File in the SportsStore Folder

```
...
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  "project": {
    "name": "sports-store",
    "ejected": true
  },
  "apps": [
    {
      "root": "ClientApp",
      "outDir": "dist",
      "assets": [
        "assets",
        "favicon.ico"
      ],
...
```

Run the command shown in Listing 3-6 in the `SportsStore` folder to create a webpack configuration file that can be used to build and run the project, a process known as *ejecting* the project from @angular/cli.

Listing 3-6. Ejecting the Project

```
ng eject
```

The ejection process updates the `package.json` file, which NPM uses to keep track of the packages used by the project. In some cases, the ejection process will add additional NPM

packages to the `project.json` file, so run the command shown in Listing 3-7 to ensure that any new packages are downloaded and installed.

Listing 3-7. Updating the NPM Packages

```
npm install
```

Creating the ASP.NET Core MVC Part of the Project

Once the Angular project has been set up, the next step is to create an MVC project in the same `SportsStore` folder. Run the command shown in Listing 3-8 in the `SportsStore` folder to create a basic MVC project.

Listing 3-8. Creating the ASP.NET Core MVC Project

```
dotnet new mvc --language C# --framework netcoreapp2.0
```

Run the command shown in Listing 3-9 in the `SportsStore` folder to add a Microsoft package to the project. This NuGet package is the .NET counterpart to the NPM packages installed earlier and is used to integrate the Angular tools into Visual Studio.

Listing 3-9. Adding a NuGet Package to the Project

```
dotnet add package Microsoft.AspNetCore.SpaServices
```

The application will be stored using Microsoft SQL Server and accessed using Entity Framework Core. The meta-package that sets up ASP.NET Core includes the main Entity Framework Core packages but there is one addition that must be made manually.

Open the project using Visual Studio or Visual Studio Code. If you are using Visual Studio, then select File > Open > Project/Solution, navigate to the `SportsStore` folder, and select the `SportsStore.csproj` file. To edit the NuGet packages, right-click the SportsStore project item in the Solution Explorer, select Edit SportsStore.csproj from the pop-up menu, and make the changes shown in Listing 3-10.

If you are using Visual Studio Code, open the SportsStore project, click `SportsStore.csproj` in the file list to open the file for editing, and make the changes shown in Listing 3-10.

Listing 3-10. Adding a NuGet Package in the SportsStore.csproj File in the SportsStore Folder

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.5" />
    <PackageReference Include="Microsoft.AspNetCore.SpaServices" Version="2.0.2" />
  </ItemGroup>

  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools"
        Version="2.0.2" />
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
        Version="2.0.1"  />
  </ItemGroup>

</Project>
```

This package provides the command-line tools for working with Entity Framework Core.

If you are using Visual Studio, the new packages will be downloaded when you save the `SportsStore.csproj` file. If you are using Visual Studio Code, use a command prompt to run the command shown in Listing 3-11 in the `SportsStore` folder.

Listing 3-11. Restoring Packages

```
dotnet restore
```

Configuring the Project

Now that the project foundation is in place, it is time to configure the different parts of the application to work together.

Preparing the Project (Visual Studio)

If you are using Visual Studio, select File > Open Project/Solution, navigate to the `SportsStore` folder, and select the `SportsStore.csproj` file. Select File > Save All and save the `SportsStore.sln` file, which you can use to open the project in the future.

Select Project > SportsStore Properties, navigate to the Debug section of the options window, and ensure that IIS Express is selected for the Launch field, as shown in Figure 3-1.

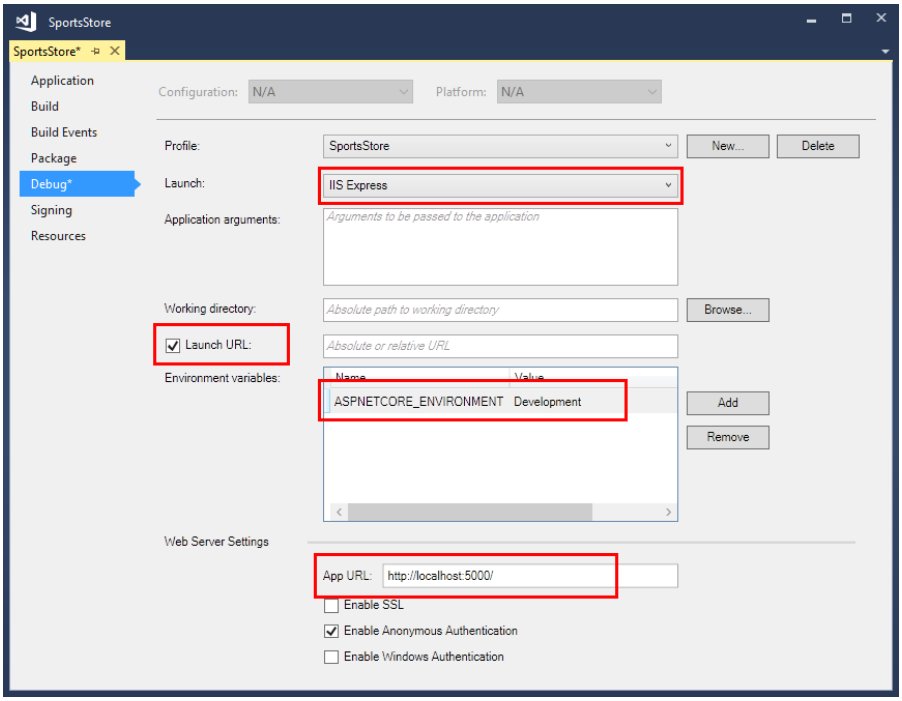

Figure 3-1. Configuring the application

Ensure that the Launch URL button is selected and enter **`http://localhost:5000`** in the App URL field, as shown in Figure 3-1. Finally, click the Add button to create an environment variable called **`ASPNETCORE_ENVIRONMENT`**, if it doesn't already exist, with a value of **`Development`**. Save the changes and close the properties window.

If you are using Visual Studio Code, edit the **`launchSettings.json`** file in the **`Properties`** folder to change the port for the startup URL for the SportsStore configuration item, as shown in Listing 3-12.

Listing 3-12. Configuring the Port in the launchSettings.json File in the Properties Folder

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:60574/",
      "sslPort": 0
    }
  },
```

```
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "SportsStore": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:5000/"
    }
  }
}
```

Creating and Editing the Configuration Files

Regardless of the IDE you are using, add a TypeScript file called `boot.ts` to the `SportsStore/ClientApp` folder, with the code shown in Listing 3-13.

Listing 3-13. The Contents of the boot.ts File in the ClientApp Folder

```
import { enableProdMode } from "@angular/core";
import { platformBrowserDynamic } from "@angular/platform-browser-dynamic";
import { AppModule } from "./app/app.module";

const bootApplication = () => {
    platformBrowserDynamic().bootstrapModule(AppModule);
};

if (module["hot"]) {
    module["hot"].accept();
    module["hot"].dispose(() => {
        const oldRootElem = document.querySelector("app-root");
        const newRootElem = document.createElement("app-root");
        oldRootElem.parentNode.insertBefore(newRootElem, oldRootElem);
        platformBrowserDynamic().destroy();
    });
}

if (document.readyState === "complete") {
    bootApplication();
```

```
} else {
    document.addEventListener("DOMContentLoaded", bootApplication);
}
```

This file is responsible for loading the Angular application and responding to changes to the client-side code. Next, edit the `Startup.cs` file to change the code in the `Configure` method, as shown in Listing 3-14. The additions enable the integration between the ASP.NET Core and Angular development tools.

Listing 3-14. Enabling Middleware in the Startup.cs File in the SportsStore Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.SpaServices.Webpack;

namespace SportsStore {
  public class Startup {
    public Startup(IConfiguration configuration) {
      Configuration = configuration;
    }

    public IConfiguration Configuration { get; }


    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {

      app.UseDeveloperExceptionPage();
      app.UseWebpackDevMiddleware(new WebpackDevMiddlewareOptions {
          HotModuleReplacement = true
      });

      //if (env.IsDevelopment()) {
      //  app.UseDeveloperExceptionPage();
      //} else {
      //  app.UseExceptionHandler("/Home/Error");
      //}

      app.UseStaticFiles();
```

```
      app.UseMvc(routes => {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
      });
    }
  }
}
```

Open the `webpack.config.js` file in the `SportsStore` folder, locate the `module.exports` statement shown in Listing 3-15, and make the changes in bold. This file contains a lot of configuration statements and the ones you are looking for follow the long list of entries that deal with "rxjs" files.

Listing 3-15. Setting the Public Path in the webpack.config.js File in the SportsStore Folder

```
...
"resolveLoader": {
  "modules": [
    "./node_modules",
    "./node_modules"
  ]
},
"entry": {
  "main": [
    "./ClientApp\\boot.ts"
  ],
  "polyfills": [
    "./ClientApp\\polyfills.ts"
  ],
  "styles": [
    "./ClientApp\\styles.css",
    "./wwwroot/lib/bootstrap/dist/css/bootstrap.min.css"
  ]
},
"output": {
  "path": path.join(process.cwd(), "wwwroot/dist"),
  "filename": "[name].bundle.js",
  "chunkFilename": "[id].chunk.js",
  "crossOriginLoading": false,
  "publicPath": "/app/"
},
...
```

Enabling Logging Messages

During development, it can be useful to see details of the build process for the Angular application and details of the queries sent to the database by Entity Framework Core. To enable logging messages, add the configuration statements shown in Listing 3-16 to the `appsettings.json` file.

Listing 3-16. Enabling Logging in the appsettings.json File in the SportsStore Folder

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Microsoft.EntityFrameworkCore": "Information",
      "Microsoft.AspNetCore.NodeServices": "Information",
      "Default": "Warning"
    }
  }
}
```

Installing the Bootstrap Package

Throughout this book, I use the Bootstrap CSS package to style the HTML elements displayed by the browser. Client-side packages are managed Bower. To configure Bower so that it installs its packages in the right place for ASP.NET Core MVC projects, add a file called `.bowerrc` to the SportsStore folder with the content shown in Listing 3-17.

Tip If you are using Visual Studio, you can add both of the files required to set up Bower using the Bower Configuration File item template.

Listing 3-17. The Contents of the .bowerrc File in the SportsStore Folder

```
{
  "directory": "wwwroot/lib"
}
```

Next, add a file called `bower.json` to the `SportsStore` folder with the contents shown in Listing 3-18.

Listing 3-18. The Contents of the bower.json file in the SportsStore Folder

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0"
  }
}
```

If you are using Visual Studio Code, use a command prompt to run the command shown in Listing 3-19 to download the Bootstrap package. (This is done automatically by Visual Studio.)

Listing 3-19. Updating the Client-Side Packages

```
bower install
```

Removing Files

Some files in the project can be removed. Table 3-2 describes the files that are not required in this book. You don't have to remove these files, but it will help simplify the project structure and, in the case of the `.editorconfig` file, avoid confusion when your code editor doesn't respect your normal preferences.

Table 3-2. *Files That Can Be Removed*

Name	Description
`.editorconfig`	This file contains project-specific editor configuration settings, and it is used by Visual Studio to override the preferences specified by the Tools > Options menu, including setting the tab size to two spaces.
`e2e`	This folder contains tests for Protractor, which does end-to-end testing for Angular applications. See `www.protractortest.org` for details.
`protractor.conf.js`	This file contains configuration settings for Protractor.
`README.md`	This file contains welcome text containing an overview of the @angular/cli tools.

Updating the Controller, Layout, and View

The final step in setting up the project is to update the controller and the Razor layout and view to replace the placeholder content and incorporate the Angular application. Edit the `Home` controller and replace the contents with the code shown in Listing 3-20.

Listing 3-20. Replacing Content in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;

namespace SportsStore.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            return View();
        }
    }
}
```

Edit the `_Layout.cshtml` file in the `Views/Shared` folder and replace the content with the elements shown in Listing 3-21.

Listing 3-21. Replacing Content in the _Layout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>SportsStore @ViewData["Title"]</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
</head>
<body>
    @RenderBody()
    @RenderSection("Scripts", required: false)
</body>
</html>
```

Edit the `Index.cshtml` file in the `Views/Home` folder and replace the contents with those shown in Listing 3-22.

Listing 3-22. Replacing Content in the Index.cshtml File in the Views/Home Folder

```
@section scripts {
    <script src="~/dist/inline.bundle.js" asp-append-version="true"></script>
    <script src="~/dist/polyfills.bundle.js" asp-append-version="true"></script>
    <script src="~/dist/vendor.bundle.js" asp-append-version="true"></script>
```

```
    <script src="~/dist/main.bundle.js" asp-append-version="true"></script>
}

<div class="navbar navbar-dark bg-dark">
    <a class="navbar-brand" href="#">@(ViewBag.Message ?? "SPORTS STORE")</a>
</div>
<div class="p-1">
    <app-root></app-root>
</div>
```

The `script` elements in this view include bundles of JavaScript files that contain the Angular framework and the client-side application code. These bundles will be created automatically by webpack when the files in the Angular part of the project change.

The `app-root` element will be replaced with dynamic content produced by the Angular application, as you will see in the next section. The other elements are standard HTML that has been styled with Bootstrap.

Running the Project

If you are using Visual Studio, then start the application by selecting Start Without Debugging from the Debug menu. A new browser window will open and request `http://localhost:5000`, which was the URL you used to configure the project earlier in the chapter, showing the content in Figure 3-2.

If you are using Visual Studio Code, then use a command window to run the command shown in Listing 3-23 in the `SportsStore` folder, which will build the project and start the ASP.NET Core HTTP server.

Listing 3-23. Starting the Project

```
dotnet run
```

Open a new browser window and navigate to `http://localhost:5000`. If you have followed all of the configuration steps, you will see the content shown in Figure 3-2.

Tip It can take a moment for the application to deliver the content to the client for the first HTTP request. Reload the browser page if you just see the SportsStore header but not the "Welcome to app!" message or just an empty page.

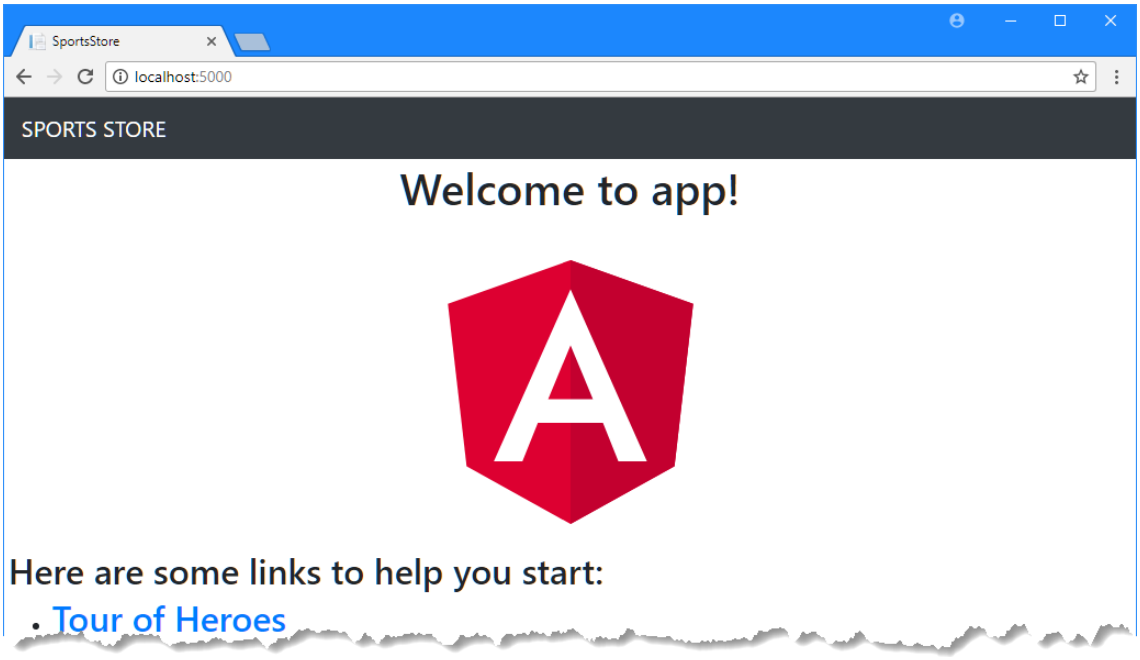

Figure 3-2. Testing the application

This may not seem like an impressive result, but the Angular development tools and ASP.NET Core are now working together. Edit the `app.component.ts` file in the `SportsStore/ClientApp/app` folder and make the change highlighted in Listing 3-24. (You will have to expand the `app.component.html` file to see the `app.component.ts` file in the Solution Explorer if you are using Visual Studio, which nests related files together.)

Listing 3-24. Making a Change in the app.component.ts File in the SportsStore/ClientApp/app Folder

```
import { Component } from '@angular/core';

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})
export class AppComponent {
    title = 'Angular & ASP.NET Core MVC';
}
```

As soon as you save the file, the Angular development tools will compile the TypeScript files and generate new JavaScript files that will be loaded and executed by the browser. The

changed files are sent to the browser, and the running application is updated on the fly, producing the output shown in Figure 3-3.

Caution The automatic refresh feature is generally reliable, but you may have to reload the browser window to see the effect of some changes or after you restart the ASP.NET Core server.

Figure 3-3. Updating the application

Understanding the Combined Project and Tools

It is worth taking the time to understand how Angular and ASP.NET Core MVC have been integrated into the project, especially since they rely on different platforms.

Understanding the Structure of the Project

Combining Angular and ASP.NET Core MVC in a single project produces a complex folder structure. When working with a combined project, it is important to remember that you are

developing two distinct applications whose files happen to be collocated. Table 3-3 describes the most important parts of a combined project.

Table 3-3. *The Key Files and Folders in a Combined Project*

Name	Description
`ClientApp`	This folder contains the Angular application and its configuration files.
`ClientApp/app`	This folder contains the source code for the Angular application, including templates and stylesheets used by Angular components.
`Controllers`	This folder contains the ASP.NET Core MVC controller classes.
`Views`	This folder contains the Razor views that are rendered by the MVC controllers.
`wwwroot`	This folder contains the static content files that the project requires.
`wwwroot/dist`	This folder contains the JavaScript files produced by compiling the Angular project.
`webpack.config.js`	This file contains the configuration used to build the Angular application.

You can find the source code for the Angular application in the `ClientApp/app` folder, which is where you will create your TypeScript files, along with any related HTML templates and CSS stylesheets. Only the source code lives this folder; when the code is compiled, the files that are produced are placed into the `wwwroot/dist` folder so they can be included in the responses to HTTP requests.

You can find the source code for the ASP.NET Core MVC application in the `Controllers` and `Views` folders and in the `Models` folder, which you will create in Chapter 4 These folders contain C# and Razor files, which are used to receive HTTP requests from the client and generate responses. The rest of the files in the project are either configuration files that can be ignored for the moment or packages required to support the integration of the Angular and ASP.NET Core tools.

Understanding the Tool Integration

The key to getting Angular and ASP.NET Core to work together is the `Microsoft.AspNetCore.SpaServices` package that was added to the project earlier in the chapter. This NuGet package, which is provided by Microsoft, provides services that are useful when working with single-page application frameworks, such as Angular.

For this chapter, the most important service is the ability to run Node.js packages from inside an ASP.NET Core project, which allows the development tools used by Angular, such as webpack, to be integrated into the ASP.NET Core HTTP request pipeline.

When the first HTTP request is received by the ASP.NET Core part of the application, the middleware added to the request pipeline starts a new Node.js runtime and uses it to run webpack. Webpack compiles the TypeScript files in the `ClientApp/app` folder and creates a set of JavaScript bundles in the `wwwroot/dist` folder.

The HTTP request is handled in the usual way by the MVC framework, which directs the request to the `Index` action on the `Home` controller, which tells Razor to render the `Index.cshtml` file in the `Views/Home` folder and send the output back to the client. The HTML produced by the Razor view contains `script` elements that load the JavaScript bundles created by webpack, which load the Angular application and display the message in the browser, as illustrated in Figure 3-4.

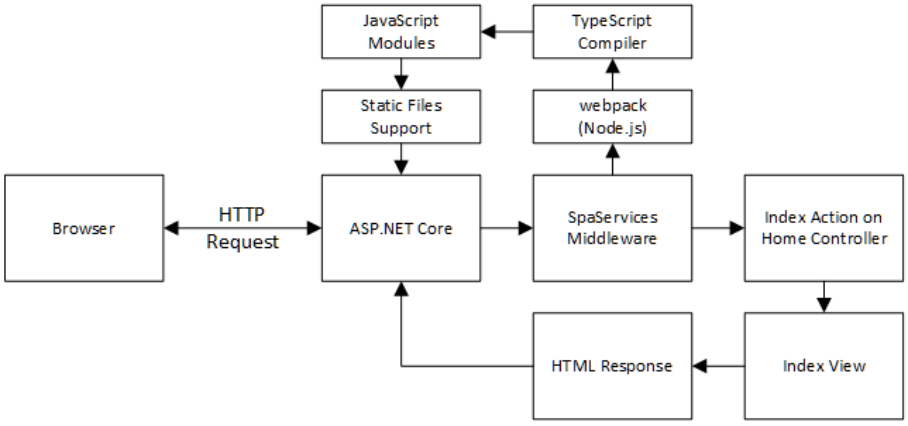

Figure 3-4. Integrating Angular and ASP.NET Core MVC

From the perspective of the MVC part of the application, the difference from a regular project is that the `SpaServices` middleware is using webpack to generate some static JavaScript files that are loaded from `script` elements included in a Razor view. From the Angular perspective, the difference from a regular project is that the HTTP requests from the browser are being handled by ASP.NET Core using Razor views, rather than through a static HTML file.

Understanding the Project Change Systems

One source of confusion is that there are three different ways in which changes to files in the projects are displayed in the browser. I describe each of these update mechanisms in the sections that follow.

Updating Angular Files

The `SpaServices` middleware is configured to support the webpack *hot module replacement* feature (known as the HMR feature), which recompiles TypeScript files and updates the running application automatically when there is a change to the code files in the Angular part of the project.

The JavaScript files created by webpack include code that opens a connection back to the server and waits for a change notification. When a file in the `ClientApp/app` folder changes, the files are recompiled, and the module files are updated. A signal is sent to the browser, which requests the modified module and uses it to update the code running in the browser without requiring a reload.

The update process was demonstrated in the "*Running the Project"* section, and it applies not just to TypeScript files but also to HTML and CSS files. Edit the `app.component.html` file to change the template used by the component in the Angular part of the project, as shown in Listing 3-25.

Listing 3-25. Editing the Template in the app.component.html File in the ClientApp/app Folder

```
<h1>
  Title: {{title}}
</h1>
```

As soon as you save the change, the Angular build process will start, and new modules will be created and used by the browser to update the client-side application without needing to reload the browser window, as illustrated by Figure 3-5.

Tip The update process can take a while when the application is first started and can, on occasion, get bogged down even when everything has been running for a while. I like to keep the browser's JavaScript console open so that I can see the HMR messages that are displayed during the update. If an update takes too long, then reloading the browser will reload the JavaScript files and give the Angular application a fresh start.

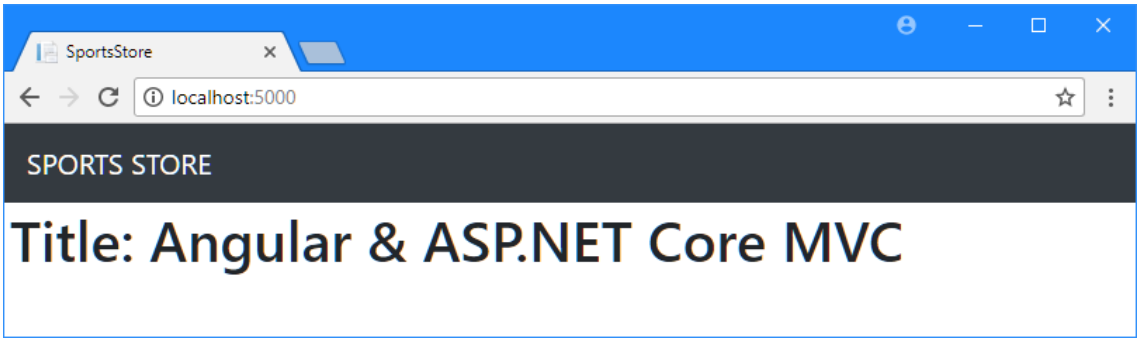

Figure 3-5. An update to an HTML template

Updating Razor Files

Razor view files are not updated as part of the HMR feature. When you make a change to a Razor view file, you must reload the browser window. The reload will send an HTTP request that will be received by ASP.NET Core, passed to the MVC framework, and used by a controller to select a view and render a new response.

Edit the `Index.cshtml` file in the `Views/Home` folder to apply the class shown in Listing 3-26.

Listing 3-26. Changing an Element Class in the Index.cshtml File in the Views/Home Folder

```
@section scripts {
    <script src="~/dist/inline.bundle.js" asp-append-version="true"></script>
    <script src="~/dist/polyfills.bundle.js" asp-append-version="true"></script>
    <script src="~/dist/vendor.bundle.js" asp-append-version="true"></script>
    <script src="~/dist/main.bundle.js" asp-append-version="true"></script>
}

<div class="navbar navbar-dark bg-dark">
    <a class="navbar-brand" href="#">@(ViewBag.Message ?? "SPORTS STORE")</a>
</div>
<div class="p-1 bg-info">
    <app-root></app-root>
</div>
```

Adding the `div` element to the `bg-info` class applies a Bootstrap CSS style that changes the background color. Nothing will happen when you save the change, but the updated view will be used if you reload the page using the browser, as shown in Figure 3-6.

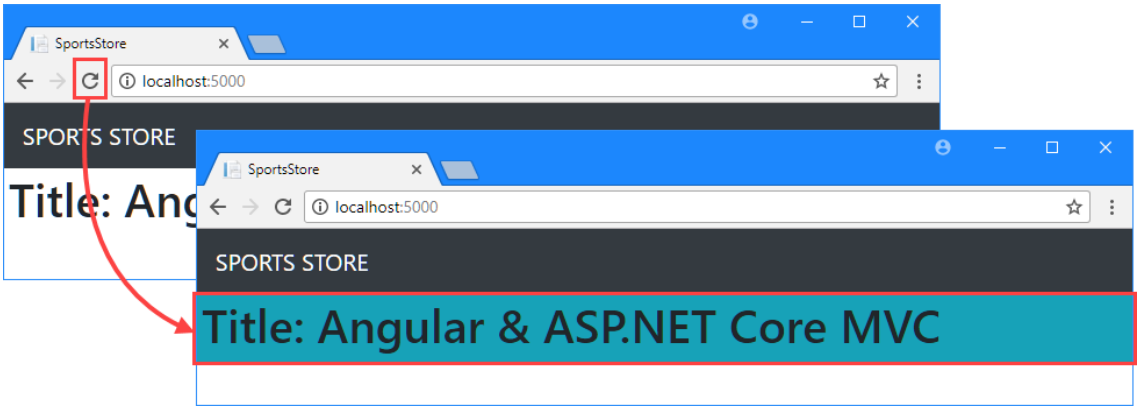

Figure 3-6. An update to a Razor view

Updating C# Classes

The effect of changes to C# classes depends on which IDE you are using. To see how changes are handled, edit the `Home` controller to change the view selected by the `Index` action, as shown in Listing 3-27.

Listing 3-27. Selecting a Different View in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;

namespace SportsStore.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            ViewBag.Message = "Sports Store App";
            return View();
        }
    }
}
```

The view bag property added to the Listing will override the text displayed at the top of the view. If you are using Visual Studio and you started the application using IIS Express by selecting Debug > Start Without Debugging, then you can trigger an update by reloading the web page in the browser. This will trigger an automatic project build and restart the application, as shown in Figure 3-7.

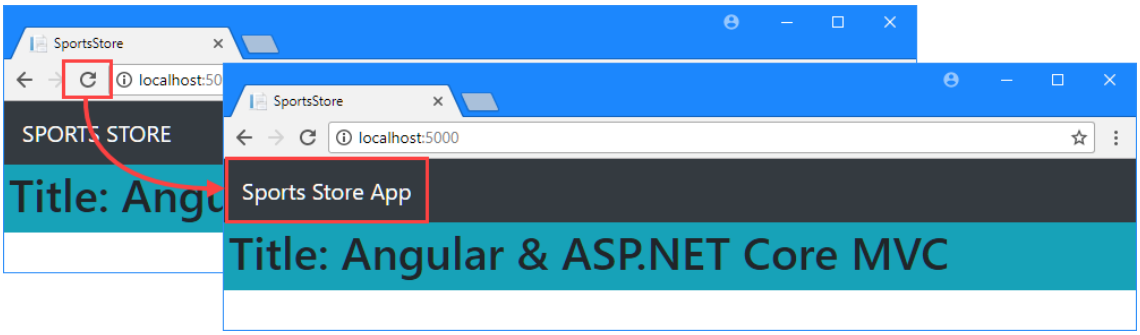

Figure 3-7. An update to an MVC controller in Visual Studio

If you are using Visual Studio Code, the `dotnet run` command used to start the application doesn't respond to changes. That means you must stop the application with Control+C and then start the application again using `dotnet run`. Once the application has started, reload the browser window to see the effect of the changes.

Tip Microsoft provides a NuGet package called DotNet Watcher, which can be used to watch the project folder for changes and trigger a rebuild and restart when a change is detected. See `https://github.com/aspnet/DotNetTools` for details.

Detecting TypeScript Errors

The hot module replacement feature will display any errors reported by the TypeScript compiler in the browser. To see an error message, add the statement shown in Listing 3-28 to the `app.component.ts` file in the `ClientApp/app` folder.

Listing 3-28. Forcing an Error in the app.component.ts File in the ClientApp/app Folder

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
    title = 'Angular & ASP.NET Core MVC';
```

```
    title = no_such_object;
}
```

The statement added to the TypeScript component class creates two errors by assigning a nonexistent value to a property that has already been defined. As soon as you save the file, the Angular development tools will compile the class, regenerate the JavaScript modules, and try to do a hot replacement. But since the code can't be executed, you will see error messages instead:

```
...
fail: Microsoft.AspNetCore.NodeServices[0]
      ERROR in ClientApp/app/app.component.ts(11,3): error TS2300: Duplicate
          identifier 'title'.
fail: Microsoft.AspNetCore.NodeServices[0]
      ClientApp/app/app.component.ts(11,11): error TS2304: Cannot find name
          'no_such_object'.
...
```

For errors like the one introduced in Listing 3-28, you will also see a warning displayed by IntelliSense in Visual Studio or Visual Studio Code, as shown in Figure 3-9. (This shows Visual Studio Code but with the color scheme changed to Light to make the text easier to see on the page.)

Figure 3-8. A code error detected by the code editor

Visual Studio and Visual Studio Code both compile TypeScript files automatically to detect and report errors in the code editor. This won't always display errors that are not simple code mistakes, so you must get used to handling errors displayed in the browser window, too.

If you are using Visual Studio, you can also select Build > Build Solution, which will compile all of the files in the project, including TypeScript files, and problems are reported in the Error List window, as shown in Figure 3-10.

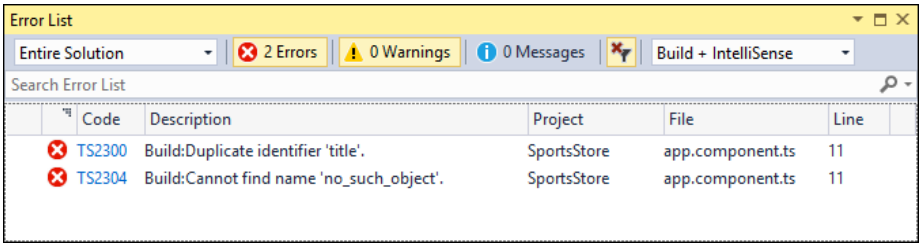

Figure 3-9. Build errors reported by Visual Studio

To restore the application to its working state, comment out the problem statement, as shown in Listing 3-29.

Listing 3-29. Commenting Out a Statement in the app.component.ts File in the ClientApp/app Folder

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
    title = 'Angular & ASP.NET Core MVC';

    //title = no_such_object;
}
```

As soon as you save the change, the TypeScript file will be recompiled and used to create a module that will be used by the code running the browser to update the application, as shown in Figure 3-11.

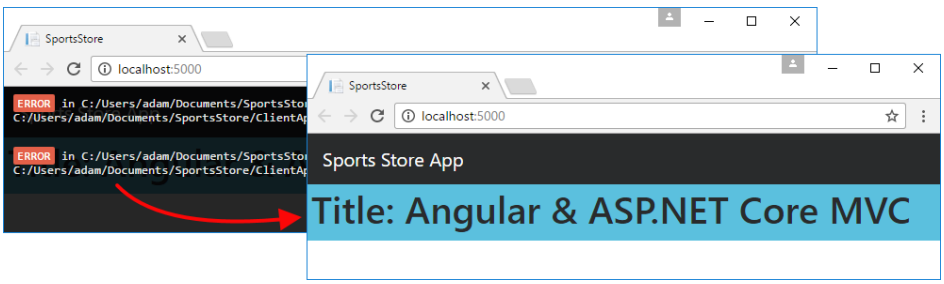

Figure 3-11. Correcting an error

Visual Studio doesn't always manage to recover from errors automatically if you have selected Build > Build Solution. Select Debug > Start Without Debugging to restart the .NET Core runtime.

Summary

In this chapter, I showed you how to create a project that combines Angular and ASP.NET Core MVC. The process is a little complicated, but the result is a solid foundation that simplifies the development process and makes it easier to get the Angular and MVC parts of an application to work together. In the next chapter, I start work on the data model, which will underpin both the ASP.NET Core MVC and Angular parts of the SportsStore project.